METHOD AND APPARATUS FOR VERSIONING STATICALLY BOUND FILES

CROSS-REFERENCE TO RELATED APPLICATIONS

FIELD OF THE INVENTION

This invention relates generally to the management of component files used in the compilation of an executable program and more specifically to the management of version information for library files that are statically bound in the compilation of the executable program.

DESCRIPTION OF THE RELATED ART

Software developers are often faced with the task of constructing new builds of software systems that involve sometimes hundreds or even thousands of component files. For building software systems in the UNIX or UNIX-like environment, it is common to construct a software system object file (the final executable file) by an invocation of the C compiler in which a string variable specifies a long list of the component files that must be compiled or linked to create the executable. Many of the component files used to build the software system are kept in one or more libraries and it is common for several versions of these libraries to be present and available to support the build process for multiple software systems. Often a particular software system requires a version of a library that was developed especially for the particular software system.

A common way of building a software system is to compile (as used herein the term compiling includes linking and may cause only linking to occur if only object files are specified) the files of one or more libraries directly into the final executable file for the system (otherwise known as statically binding the files that make up the final executable file). However, once the final executable file has been created, it is difficult to identify exactly which version of the library was used to construct the system. Out-of-date or incorrect versions of these libraries may cause the system to behave incorrectly, often in mysterious ways. Also, software support personal may want to verify that the version of a library that was used in a build matches the latest released version of the library.

Thus, there is a need for a method of identifying the version of one or more libraries that are used to build a software system to help determine whether the versions of the one or more libraries used were correct for that software system.

BRIEF SUMMARY OF THE INVENTION

The present invention is directed towards such a need. A computerized method of saving version and product information of a library in an executable program, in accordance with the present invention, includes creating a version source file for the library, where the version source file contains version and product information pertaining to the library. Next, the version source file is compiled to create a version object file, following which the library is rebuilt to include the version object file. The executable program is then built so as to include the version object file of the library such that the version and product information is combined into the executable program.

In one embodiment, a compound library is formed from a selected group of libraries, each of which has a version object file. The compound library contains the object files of the selected group, including the version object files of each library in the selected group, and a version object file for the compound library. When the compound library is included in the build of the executable, the executable has the version information for each library in the selected group and the version information for the compound library itself.

One advantage of the present invention is that it is possible to identify the version of a library that was used to build a software system and whether the correct library was used for the build.

Another advantage is that subtle software errors that are caused by using an outdated or incorrect library can be traced.

Yet another advantage is that software releases can be made more reliable because the component files used in the build can be checked prior to release to assure that they are the correct and up-to-date component files.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features, aspects and advantages of the present invention will become better understood with regard to the following description, appended claims, and accompanying drawings where:

FIG. 1 shows a typical computer system for storing and executing the processes of the present invention;

FIG. 2 shows a first library file (libX.a) that comprises three object files, one of which is the version information file for the first library;

FIG. 3 shows a second library file (libY.a) that comprises three object files, one of which is a version information file for the second library;

FIG. 4 shows an executable object file for a particular software system which is built with the first and second libraries and includes the version information for the first and second libraries as well as the version information for the software system itself;

FIG. 5 shows the listing after running a utility program on the executable object file of FIG. 4 to list the component object files used to create the executable object file;

FIG. 6 shows a flow chart for creating and storing the version information file for a library;

FIG. 7 shows a flow chart for building a software system in which the versioning information from each of the libraries is accessed and included in the build;

FIG. 8 shows a flow chart for building an executable to include a compound library; and

FIG. 9 shows a flow chart for creating a compound library.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. 1 shows a typical computer system for storing and executing the processes of the present invention. Central processing unit 2, main memory 4, storage controller 6, I/O controller 8, and communications controller 14 are interconnected via system bus 16. Main memory 4 contains programs which are executed by central processing unit 2. The storage controller 6 transfers programs and data from external storage devices to and from main memory 4. External storage devices may hold libraries, compilers and other programs that pertain to the present invention. Libraries and other programs are brought in to the memory as they are needed by a currently executing program. The I/O controller 8 couples information from mouse and keyboard devices to the central processing unit 4 and communications I/O 14 connects the central processing unit and the main memory to an external communications medium such as a local area network.

FIG. 2 shows a first library file (libX.a) 10 that comprises three object files, one of which is the version information file for the first library. The first object file, X_1.o, in the library contains two functions, `ftn_X_1_1()` and `ftn_X_1_2()`. The second object file, X_2.o, contains one function, `ftn_X_2()`, and the third object file, vproc_libX.o, contains a version function 12, `T1111V01_30sep2000_08jul2000_X()`, to indicate that library libX.a is part of product T1111, version V01, having a customer release date of September 30, 2000 and a build date of July 8, 2000. A utility (such as an archive command, ar –t libX.a, in the UNIX environment) can be used to list these three component object files of the libX.a library.

FIG. 3 shows a second library file (libY.a) 20 that comprises three object files, one of which is a version information file for the second library. File Y_1.o, the first object file, contains function `ftn_Y_1()`. The second object file, Y_2.o, in the library contains two functions `ftn_Y_2_1()` and `ftn_Y_2_2()`. The third file object file, vproc_libY.o, contains a version function 22, `T2222V02_30sep2000_19jul2000_Y()`, to indicate that library libY.a is part of product T2222, version V02, having a customer release date of September 30, 2000 and a build date of July 19, 2000. A command (such as an archive command, ar –t libY.a, in the UNIX environment) can be used to list these three component object files of the libY.a library.

FIG. 4 shows an executable object file 30 for a particular software system which is built with the first and second libraries and includes the version information for the first and second libraries as well as the version information for the software system itself. The executable object file 30 includes the functions from each of the libraries, libX.a and libY.a, and each of the version functions 12, 22 from those libraries, along with a version function 32, `T0123V01_30sep2000_20jul2000()`, for the executable object file itself.

FIG. 5 shows the listing 40 after running a utility program on the executable object file of FIG. 4. When the utility program is run with the executable object file as an argument, it lists the version functions for each of the libraries and each additional version function or procedure contained in the file. Thus, if the utility program is run on the object file of FIG. 3, the relevant output is:

```
Version procedure:  T0123V01_30SEP2000_20JUL2000
Version procedure:  T1111V01_30SEP2000_09JUL2000_X
Version procedure:  T2222V02_30SEP2000_19JUL2000_Y
Version procedure:  T8432D44_01DEC1997_CRTLMAIN
```

The first line of the output indicates the version information for the executable object file, while the second and third lines indicate the version information for the library files, libX.a and libY.a. The last line indicates the version of function `main()` from the C Runtime Library that was statically linked into the executable. This information allows one to determine which versions of the component files were used to construct the executable object file and whether the correct component files were used because the product and version information and the build identifier are included in the version functions. In some embodiments, only the product and version information without the build identifier are included.

To build an executable having version functions, two major steps are required. First, a version function must be created and stored into each respective library in a way that makes finding that version function possible. Second, when the library is used in a compilation to create a final executable, the version function for that library must be obtained and included in the compilation. In some embodiments, a compound library is formed from two or more libraries, each of which has a version object file. In addition the compound library has its own version object file. These steps make it possible to determine the particular libraries that were used in compiling a particular final executable by examining the version functions stored in the executable.

FIG 6 shows a flow chart for the first major step in accordance with a preferred embodiment of the present invention. First, in step 52, a version string is formed, for example, T1111V01_30sep2000. Next, in step 54, a build identifier is combined with the version string and function symbols to convert the string into a build-indicated, version function, (a void function with an empty body is preferred but not essential to the invention; a non-void and/or non-empty function with identifying strings or text in its body is within the scope of the present invention). In the preferred embodiment, the build identifier (shown as a build date) is appended to the version string and the function symbols are appended to the resulting string to form the build-identified version function. The build identifier, in step 54, includes any mark that can serve to distinguish one build from another build. Preferably, the build identifier is a build date character string, but a unique build number or a user id can serve equally well as a distinguishing mark. In some embodiments a build identifier is not appended to the version string. In step 56, a version source file is created. This source file, vproc_libX.c, contains the build-indicated version function (or just the version function, if no build identifier is used) and has a name, vproc_libX, that is formed from the combination of a keyword ('vproc' in this case) and the name of the library (without the '.a' extension) at issue, 'libX'. In the preferred embodiment, the library name is appended to an underscore after the keyword. In step 58, the version source file is compiled into an object file, vproc_libX.o, having the same name (notwithstanding the file extension) as the version source file. In the directory, in step 60, any old version object file is removed from the library and a new library is made, in step 62, that includes the other object files in the library and the current version object file. This guarantees that the library is indeed consistent with the build-identified version function.

FIG. 7 shows a flow chart for the second major step. A temporary storage area, such as a temporary directory, is made, in step 70, to store extracted version object files. Next, in step 72, a list of all the libraries (libX.a and libY.a, in the example of FIG. 4) that are to be used in a new build is constructed and, in step 74, one of the libraries is selected. For that library (libY.a in the example) the version object file (vproc_libY.o) is found, in step 76, by matching the keyword, vproc, in a listing of the contents of the library. In an embodiment in which a compound library is used, all version object files in the compound library are matched. The found version object file (vproc_libY.o) is saved in the temporary storage area. In step 78, the version object file is appended to a composite string variable and, as determined in step 80, the loop continues until each of the libraries has been searched for its version object file. After the loop ends, the composite string variable contains a space separated list (vproc_libX.o vproc_libY.o) of the version object files for the libraries to be used in the build and the temporary storage area contains these files. Finally, in step 82, the compiler is called to perform the build using the composite string variable to specify all of the version object files and in step 84, the temporary storage area is deleted.

The result is that the version object files have their contents, the version functions, compiled into the build so that the executable appears, as in FIG. 4, with the version functions.

FIG. 8 shows a flow chart for building an executable to include a compound library. A compound library is a library formed from a selected group of libraries out of a plurality of libraries. Each of the plurality of libraries has a version object file that was placed in the library, in step 90, and in accordance with the flow of FIG. 6. In step 92, a group of libraries is selected to make the compound library and, in step 94, the compound library is built from the selected group. Finally, in step 96, the executable is built specifying the compound library. The result is that the executable includes the version and product information about each library in the selected group and version and product information for the compound library itself.

FIG. 9 shows a flow chart for creating a compound library. First, in step 100, a temporary storage area is created for holding the object files for each library in the selected group. Next, in step 102, all object files, including the version object files, are extracted from each library in the selected group and, in step 104, the extracted files are stored in the temporary storage area. In step 106, a version source file is constructed for the compound library, in accordance with the flow of FIG. 6, and, in step 108, the version source file for the compound library is compiled to create a version object file for the compound library and this version object file is then stored, in step 110, in the temporary storage area. In step 112, the compound library is built using the files from the temporary storage area and the compound library is saved, in step 114, in a library storage area. Finally, in step 116, the temporary storage area is deleted.

Although the present invention has been described in considerable detail with reference to certain preferred versions thereof, other versions are possible. For example, in one alternative version, the plurality of libraries includes compound and non-compound libraries such that the flow of FIG. 9 builds a multiple compound library, which includes at least one other compound library, when a compound library is in the selected group of libraries.

Therefore, the spirit and scope of the appended claims should not be limited to the description of the preferred versions contained herein.